Homework 4 Selected Solutions APPM/MATH 4650 Fall '20 Numerical Analysis

Due date: Monday, October 5, before midnight, via Gradescope. **Instructor**: Prof. Becker
Theme: Splines

solutions version 9/28/2020

Problem 1: Is the barycentric formula ill-conditioned?

a) Consider $f(x) = \frac{1}{1-x}$.

i. Find the relative condition number of evaluating f. Is this good or bad when $x \approx 1$?

Solution:

$\kappa_f(x) = \left|\frac{x}{f(x)} f'(x)\right| = \left|\frac{x}{1-x}\right|$. This is bad when $x \approx 1$.

ii. Actually evaluate f for $x = 1 - 10^{-13}$, on the computer using any software/language that uses floating point numbers[1], and report your answer and the error.

Solution:

I get `9.9969e+12`, and the correct answer is `1.0e+13`.

iii. About how many correct digits do you have? Is this expected?

Solution:

The relative error is `3.10e-4`, so about 3.5 correct digits only. This makes sense, since $\kappa(x) = 10^{13}$ so we expect to lose about 13 digits (from 16 to start with), so about 3 is right.

iv. If you evaluate f for $x = 1 - 2^{-43} \approx 1 - 1.13 \cdot 10^{-13}$, how many correct digits do you have now? (Don't forget your true answer has changed)

Solution:

I get no error at all, so all correct digits (the true answer is now 2^{42}). This is very different than before because x can now be exactly represented by floating point numbers (as can the coefficients in f), so there is no roundoff.

b) The barycentric formula for Lagrange interpolation looks something like

$$p(x) = \sum_{i=0}^{n} \frac{\frac{w_i}{x-x_i} f_i}{\frac{w_i}{x-x_i}}$$

which might make you nervous when $x \approx x_i$ for some i. Let's analyze a simplified version of this formula that still retains all the interesting (and worrisome?) behavior:

$$f(x) = \frac{\frac{c}{1-x} + d}{\frac{1}{1-x} + 1}$$

for some constant d and $c \neq 0$.

Find the relative condition number κ_f for $x = 1$.

[1]e.g., do *do not* use a symbolic computation framework like Mathematica unless you explicitly convert to floating point numbers by, e.g., writing $x = 1. - 10.^{-13.}$ where the decimal dots tell it to use floating point. Tools like graphing calculators and Wolfram Alpha are also probably to be avoided since they try to guess what you want, and could be doing it in either exact arithmetic or floating point.

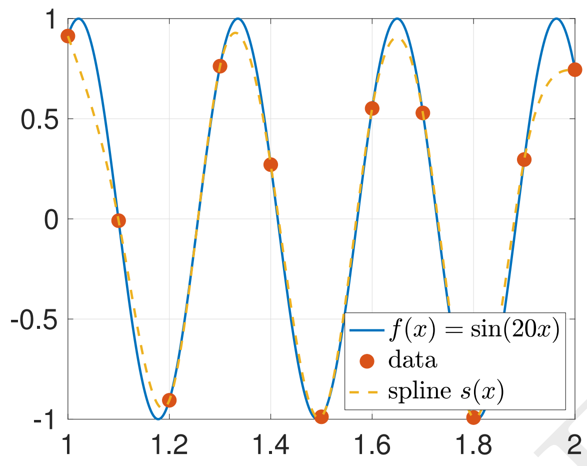


Figure 1: Example of spline for Problem 2

Solution:

It's probably easiest to first multiply by $1-x$. We find $\kappa_f(1) = \boxed{\left|\frac{c-d}{c}\right|}$ so for c not close to 0, this is not ill-conditioned at all.

It's not necessary for the homework but we could verify that we get high accuracy with this formula. Let's say $c=2$ and $d=2$, and let $x = 1-10^{-13}$, so then $f(x) = \frac{2\cdot 10^{13}+2}{1^{13}+1} = 2$, and numerically evaluating f on the computer gives *exactly* 2, so we have perfect precision even though x was inaccurate!

Problem 2: **Splines** For this problem, we will do cubic splines. You do not need to implement these yourselves; in Matlab, you can use the curve-fitting toolbox (should be free with our campus subscription; you can also remotely use CU desktops if you need to) and the functions `csape` (to find the spline) and `fnval` (to evaluate the spline); in Python, use `scipy.interpolate.CubicSpline`.

We'll use the function

$$f(x) = \sin(20x)$$

and the interval $[1,2]$. This function will give us the values to use on the nodes. We will only consider equispaced nodes $[x_0 = 1, x_1 = 1+h, \ldots, x_n = 1+nh = 2]$ for $h = 1/n$, and will choose different values for n.

a) Create both the **natural** cubic spline as well as the **not-a-knot** cubic spline (using the library code; you do NOT need to program the details yourself). Denote the spline by $s(x)$. We want to measure the error $|s(x) - f(x)|$ for generic points x. For this sub-problem, sample 10^5 points uniformly at random from the interval $[1.01, 1.99]$ (use Matlab's `rand` or Python's `numpy.random.rand`), and report the maximum error $|s(x) - f(x)|$ for these x values. Plot this error, for both types of cubic splines, on a figure as a function of how many points n (or $n+1$) are used for nodes. Use the plot to graphically demonstrate the order of convergence of the error. *Hint*: Make sure you have a wide enough range of n values to make this convincing and make wise choices about whether axes should be linear or logarithmic.

Solution:

We get the following plot. This should be a log-log plot, and cover enough range of n (I chose 4 orders of magnitude) to show the true behavior. We plot guide lines of $cn^{-\alpha}$ for various exponents α in order to find the slope; the constant c can be adjusted to raise/lower these lines in order to make it more clear.

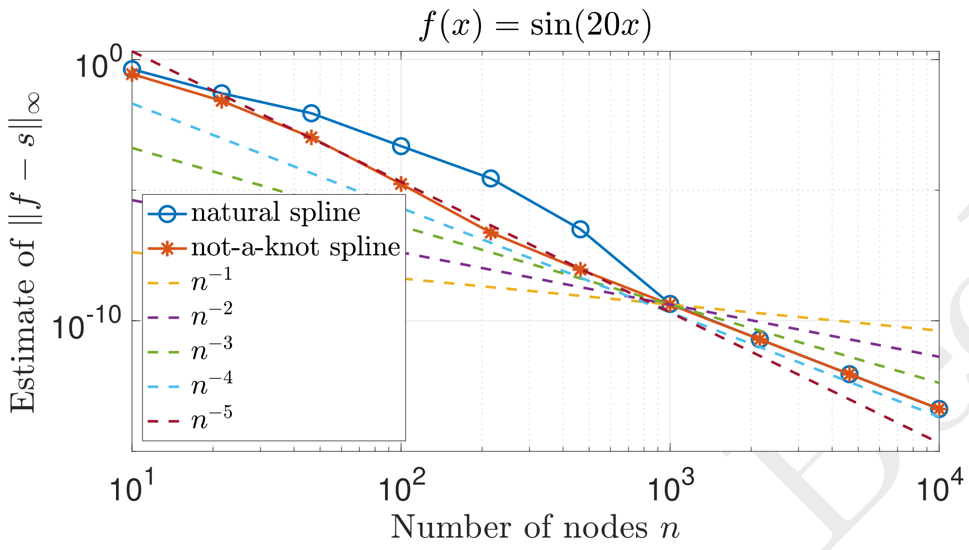


Figure 2: Problem 2a. For $n \geq 10^3$, the n^{-4} line is the best fit. For smaller n, n^{-5} might look good, but we ultimately care about asymptotic behavior (large n)

.

Thus we seem from the graph in Fig. 2 that for large enough n, both types of splines have $\boxed{O(n^{-4}) = O(h^4) \text{ convergence}}$.

Comment 1: Many students may be tempted to apply a linear fit to the data (well, to log-transformed data) using `polyfit` or similar, as this gives a "precise" measure of the rate. However, this extra precision (say, finding a rate of $h^{4.0231}$) is not important, and it hides the fact that we care much more about "goodness of fit" rather than a very precise value of the slope. That is, we need a small residual (or "RSS"), and since this is log-transformed data, you'd have to be careful how to interpret what a "good" value of RSS is. Most importantly, we usually don't see the error settle down into the asymptotic h^k behavior until h is quite small, so if you apply a linear fit to all the data, then you are fitting to this transient behavior. So if you do a linear fit of the data, you should only apply it to the data after you're certain its in the asymptotic behavior regime. And even if you do this, you should still plot the rate you see, as this lets other visually confirm how accurate your fit is. So bottom line is that for this kind of problem, a good figure is the gold-standard. Furthermore, we usually plot several slopes, as this gives the viewer a quick way to gauge the error in your estimate.

Comment 2: Do not confuse this rate of convergence with the "order of convergence" α, where $\alpha = 1$ is linear convergence and $\alpha = 2$ is quadratic convergence. In terms of "order of convergence", what we're seeing is all *sublinear*, and we're trying to figure out what kind of sublinear convergence we have, e.g., is it h^2 or h^3, etc.? (Think of $h = 1/n$, and so we're seeing if we have $e_n = 1/n^2$ or $e_n = 1/n^3$, etc.).

b) Is the order of convergence you found above expected? Discuss briefly

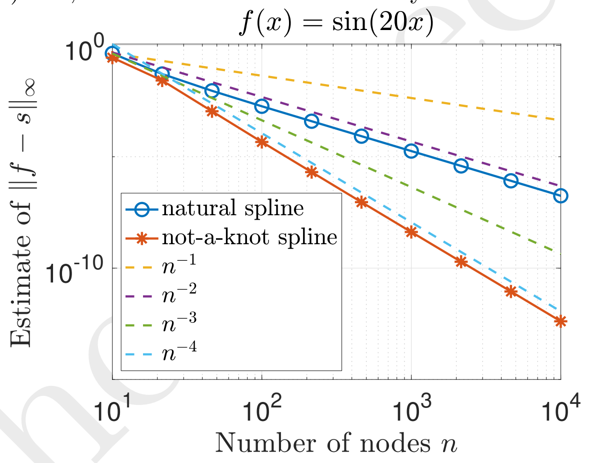
Solution:

Yes, this agrees with the fact that $f \in C^2[0,1]$ so Theorem 3.13 applies (which is for clamped splines, but similar theorems hold for natural and not-a-knot splines).

c) Repeat problem (a) but this time sample the 10^5 test points uniformly from $[1,2]$, and show the same kind of plot as in (a). Are the results the same or different? Discuss briefly.

Solution:

The results are now different, as the natural spline appears to converge at just $O(n^{-2})$ now, though the not-a-knot spline still converges at $O(n^{-4})$. This is probably due to the fact that Theorem 3.13 does not hold exactly for natural splines; rather, we get $O(h^4)$ convergence for any *fixed* $x \in (0,1)$, and what we're seeing is that the worst points are near the boundary, and we've sampled so many test points that we have these bad points near the boundary. But if we don't increase the number of test points, then at some point when n is large enough, we will get $O(h^4)$ convergence. Basically, we're seeing the behavior we saw in part (a) for $n < 500$, but now the behavior extends past $n = 10^4$. To get uniform convergence on $[0,1]$ we would need $f''(1) = f''(2) = 0$, since otherwise these directly conflict with the natural spline boundary conditions.



Comment: what is *uniform* convergence? In our case, it is convergence of the quantity $\sup_{x\in[1,2]} |s_n(x) - f(x)|$, e.g., that $\sup_{x\in[1,2]} |s_n(x) - f(x)| = O(n^{-4})$. This is in contrast to *pointwise* convergence which is that $\forall x \in [1,2]$ that $|s_n(x) - f(x)| \to 0$, e.g., $|s_n(x) - f(x)| = O(n^{-4})$. Uniform convergence implies pointwise convergence and is the stronger notion. In both cases, we converge at any point x, but for uniform, we can give a uniform rate for all x. What happens with the natural spline case is that the worst-case x changes as n increases.

d) Repeat problem (a), but get the values from the function g, where

$$g(x) = \begin{cases} f(x) & x < 1.3 \\ f(2.6 - x) & x \geq 1.3 \end{cases}.$$

Show the same kind of plot as before. Are the results the same or different? Discuss briefly.

Solution:

Both methods converge at about $O(n^{-1}) = O(h^1)$. We don't expect fast $O(h^4)$ convergence since the function $g \notin C^4[1,2]$ since it is not differentiable at $x = 1.3$, and since it is not in C^4 we cannot apply Theorem 3.13. Looking at the figure, we seem to have about $O(n^{-1})$ convergence mostly, though it does get faster as $n > 10^6$. We're not looking for a precise number here, just observing that the lack of smoothness slows down convergence.

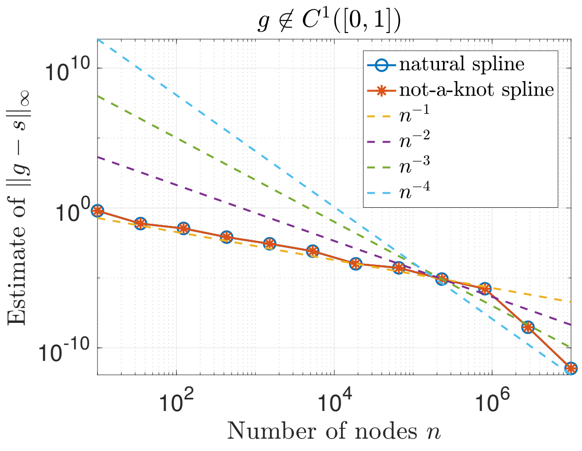
Note: to implement g efficiently, it's best to avoid any kind of `if` statement, since then you cannot vectorize it. In Matlab, an efficient implementation is

```
g   = @(x) (x<1.3).*f(x) + (x>=1.3).*f( 2*1.3-x );
```

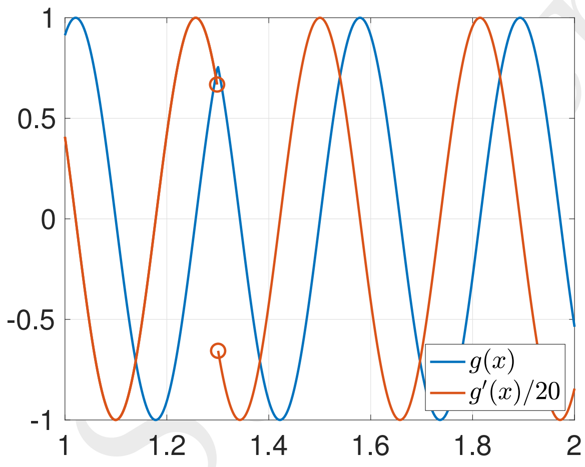
and in Python (this will work well with `np.array`):

```
import numpy as np
f   = lambda x : np.sin(20*x);
g   = lambda x : (x<1.3)*f(x) + (x>=1.3)*f( 2*1.3-x );
```

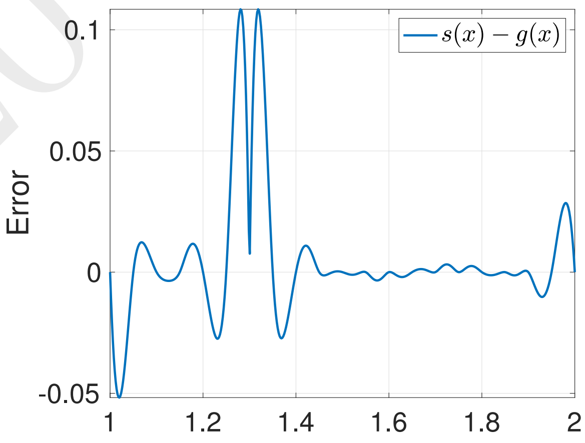
Also, in Matlab please use `linspace` and similar to make the nodes (e.g., `nodes = linspace(a,b,n+1);`); in Python use `numpy.linspace` or `numpy.arange`. Do not use `for` loops, as these are super slow!



We can see that it is non-smooth by plotting it:



and notice that the largest error by the spline is made near the point of discontinuity at $x = 1.3$:



Note: in Python, there are other ways to implement g, including `np.piecewise`, though this may not be as efficient as the method I suggested above. For example, here is some timing code that I ran in colab: (we use `numpy.vectorize` so we can automatically loop over a `numpy.array` object, and we're using this in in its decorator form).

```python
import numpy as np
from timeit import timeit

f = lambda x : np.sin(20*x)
n = 1e7;
x = np.linspace(1,2,int(n))

@np.vectorize  # This tells Python to apply np.vectorize to the following
    function
def g_slow(x):
    if x < 1.3:
        return f(x)
    else:
        return f(2.6-x)
g1 = lambda x : (x<1.3)*f(x) + (x>=1.3)*f(2.6-x)
g2 = lambda x : np.piecewise( x, x<1.3, [f, lambda x : f(2.6-x)] )
```

Now we'll time them. The `%` symbol is specific to iPython (used in jupyter) and is a line magic.

```python
%time g_slow(x)
```

which gives 12.1 seconds;

```python
%time g1(x)
```

which gives 0.5 seconds; and

```python
%time g2(x)
```

which gives 4.0 seconds. So the `np.piecewise` is better than just doing `numpy.vectorize`, but implementing it as in `g1` is by far the fastest (even though we do extra work).